New Parallel Algorithms for Finding Determinants of N × N Matrices

Sami Almalki, Saeed Alzahrani, Abdullatif Alabdullatif
College of Computer and Information Sciences
King Saud University
Riyadh, Saudi Arabia
samifx@gmail.com, eng.analyst@yahoo.com, amalabdullatif@ksu.edu.sa

ABSTRACT: *Determinants has been used intensively in a variety* of applications through history. It also influenced many fields of mathematics like linear algebra. Finding the determinants of a squared matrix can be done using a variety of methods, including *well-known methods of Leibniz formula and Laplace expansion which calculate the determinant of any N × N matrix in O(n!). However, decomposition methods, such as: LU decomposition, Cholesky decomposition and QR decomposition, have replaced the native methods with a significantly reduced complexity of O(n ^ 3). In this paper, we introduce two parallel algorithms for Laplace expansion and LU decomposition. Then, we analyze them and compare them with their perspective sequential algorithms in terms of run time, speed-up and efficiency, where new algorithms provided better results. At maximum, in Laplace expansion, it became 129% faster, whereas in LU Decomposition, it became 44% faster.*

Keywords: Parallel Processing, Parallel Algorithms, Laplace Equations, Linear algebra, Multithreading

Received: 29 January 2013, Revised 2 March 2013, Accepted 9 March 2013

1. Introduction

Finding the determinant value of an N x N matrix has been an important topic that is useful in many fields of mathematics. In fact, determinants have led to the study of linear algebra. Although the attitude towards determinants has been changed, but however, the theoretical importance of them can still be found. Through history, many applications have used determinants to solve some mathematical problem, for example:

- Solving linear equation systems by applying Cramer's Rule [6].
- Defining resultants and discriminants as particular determinants.
- Finding the area of a triangle, testing for collinear points, finding the equation of a line, and other geometric applications.
- Various applications in abstract mathematics, quantum chemistry, etc.

There are several methods for finding the determinant of a matrix, including the native methods of Leibniz formula and Laplace expansion (cofactor expansion) which are both of order (n!). There are also some decomposition methods, including, LU decomposition, Cholesky decomposition and QR decomposition (applied for symmetric positive definite matrices), which improve the complexity to $O(n^3)$ [3]. In addition, a special-case of Leibniz Formula is Rule of Sarrus, which applies only on

3×3 matrix, and can be thought of as a triangular memorization scheme.

Designing parallel algorithms involves using some design patterns. Those well-known patterns are implemented using parallel processing techniques, which is available in many different programming languages. The patterns are:

• **Bag of Tasks:** A number of workers maintain a dynamic bag of homogeneous tasks. All tasks are independent from each other so every worker can get a task, complete it, and return to get another one, and so on until the bag is empty.

• **Divide and Conquer:** Breaking a big task into sub-tasks, which in turn, break the sub-tasks into smaller ones until we reach the base case that can be used to solve the parent tasks. Recursion is usually used in this pattern [5]. An example for this pattern is Merge Sort.

• **Producer-Consumer:** This approach is applicable when a group of activities consumes a data produced by another group of activities. The flow of data in this pattern is always in one direction (i.e. from producers to consumers).

• **Pipeline:** The pipeline pattern is found when the consumers become producers for another group of consumers. Hence, we can say that the Producer- Consumer is a special case of the Pipeline pattern. The pipeline flow of data begins with stage 1 and continues to stage *n*. An example of this pattern is Sieve of Eratosthenes.

Lea [1] described the design and implementation of a Java framework that solves problems recursively using parallel programming by forking them into sub-tasks and then waiting them to be joined. Mahmood et al. [8] introduced a parallel algorithm to calculate the determinant of a tridiagonal matrix with a complexity of $O(\log_2 n)$. This algorithm is based on divide-and-conquer technique, and is only applicable for tridiagonal matrices. Salihu [4] introduced a new mathematical method to find determinants of $N \times N$ matrices, for $n \geq 3$, by reducing the determinants to second order.

In this paper, we are applying two methods of finding the determinant value of any $N \times N$ matrix using both sequential and parallel algorithms. The first method is Laplace expansion, which applies the divide-and-conquer pattern, and the other one is LU Decomposition.

2. Methods of Solving Determinants

2.1 Laplace expansion

Laplace expansion is a method for finding the determinant by recursively expanding each matrix to sub-matrices until reaching a base case of 2×2 matrix. It expresses a determinant $|A|$ of an $N \times N$ matrix A as the summation of all determinants of its $N{-}1 \times N{-}1$ sub-matrices.

Given an $N \times N$ matrix A, such that:

$$A = \begin{bmatrix} a_{ij} & \cdots & a_{iN} \\ \vdots & \ddots & \vdots \\ a_{Nj} & \cdots & a_{NN} \end{bmatrix}$$

where a_{ij} is the value of the element located in the *i*th row and *j*th column, and $i, j \in \{1, 2, ..., N\}$. The cofactor is then given by:

$$C_{ij} = (-1)^{i+j}.\, M_{ij}$$

where M_{ij} is the minor of A in a_{ij} . Hence, the determinant $|A|$ is then given by:

$$|A| = \sum_{j'=1}^{n} a_{ij'} C_{ij'} = \sum_{i'=1}^{n} a_{i'j} C_{i'j}$$

Depending on Laplace expansion method, we follow the following scheme to calculate $|A|$:

• Find the cofactor by multiplying the minor by either 1 if the sum of the column and row numbers is even, or -1 if the sum of the column and row numbers is odd (i.e. $(-1)^{i+j}$).

• Recursively, and for each column of a certain row, find the minor determinant of A by eliminating the elements that are in the same row and column of a_{ij} to consist a sub-matrix.

• If you reach the base case, calculate the determinant directly.

• Multiply the result by the value of a_{ij}.

• Add up all sub-determinants to calculate the total determinant of *A*.

The expansion is working in the following manner:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$|A| = a_{11} \cdot \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \cdot \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} - a_{11} \cdot \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$|A| = a_{11} \cdot (a_{22} \cdot a_{33} - a_{23} \cdot a_{32}) - a_{12} \cdot (a_{21} \cdot a_{33} - a_{23} \cdot a_{31}) + a_{11} \cdot (a_{21} \cdot a_{32} - a_{22} \cdot a_{31})$$

This method has a complexity of $O(n!)$ so it is usually not feasible to be applied in real applications. However, we will design sequential and parallel algorithms to find determinants using this method.

2.2 LU Decomposition

LU Decomposition is one of the decomposition methods for finding determinants and is a modified version of Gaussian elimination. It works by decomposing a big matrix *A* into a product of two matrices, the lower matrix *L* and the upper matrix *U*, which are simpler to calculate.

$$|A| = |L| \,.\, |U|$$

The decomposition is done by dividing the matrix A diagonally, such that, all elements above the diagonal of matrix *L* are set to 0, and all elements below the diagonal of matrix *U* are set to 0. In addition, the elements on the diagonal of matrix *L* are set to 1 to make it easier.

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{vmatrix} \cdot \begin{vmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{vmatrix}$$

Hence, in case of the previous matrix, the result will be 9 unknowns and 9 solvable equations from the product of the matrices *L* and *U*. The general formulas for *L* and *U* are:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} \cdot l_{ik}$$

$$l_{ij} = \frac{1}{u_{ij}} \left(a_{ij} - \sum_{k=1}^{j-1} u_{kj} \cdot l_{ik} \right)$$

This method, LU Decomposition is much better in both complexity and run time. It has a complexity of $O(n^3)$. We will also design parallel algorithms to find determinants using this method and will compare it with an existing sequential version of LU Decomposition.

3. Design

Borodin [7] described two approaches of designing a good parallel algorithm.The first is to start from a good sequential algorithm and build a parallel version of it. The second approach is to minimize the parallel time allowing an arbitrary number of processors. However, we have applied the first approach and started from two sequential algorithms toward the parallel version.

Our code generates a random matrix of $N \times N$. Then, it finds the determinant based on four classes that represents the four algorithms: Laplace expansion (both sequential and parallel) and LU Decomposition (both sequential and parallel).

The first sequential algorithm shown in Figure 1 uses Laplace expansion and calculates the determinant recursively in $O(n!)$. It is a direct application of finding determinants using Laplace expansion.

```
static double matrix [] [];
public static double determinant (double matrix [] [], int level)
    {
     double det = 0;
     double res;
     if (level == 1) // trivial case 1 × 1 matrix
       res = matrix [0][0];
     else if (level == 2) // base case 2 × 2 matrix
      res = matrix [0] [0] * matrix [1] [1] − matrix [1] [0] * matrix [0] [1];
     else { // N × N matrix
      res = 0;
    for (int j1 = 0; j1 < level; j1++) {
       this.matrix = new double [level− 1] [];
    for (int k = 0; k < (level − 1); k++)
      this.matrix [k] = new double [level− 1];
   for (int i = 1; i < level; i++) {
      int j2 = 0;
  for (int j = 0; j < level; j++) {
     if (j == j1)
        continue;
     this.matrix [i − 1] [ j2] = matrix [i] [ j];
    j2++;
    }
    }
   res += Math.pow(− 1.0, 1.0 + j1 + 1.0) * matrix [0] [j1]
   * determinant(this.matrix, level − 1);
   }
   }
   return res;
  }
```

Figure 1. Sequential algorithm of Laplace expansion for finding determinants

In order to parallelize the above code, we have used the Java SE fork/join framework that has been included since Java 7. Figure 2 shows our parallel version of the sequential Laplace expansion.

```
static final ForkJoinPool pool = new ForkJoinPool();
volatile double res;
final double matrix [] [];
final int level;
public DeterminantTask (final double matrix [] [], final int level)
   {
    this.matrix = matrix;
    this.level = level;
   }
   public void setRawResult(Double d) {
   this.res = d;
  }
  public Double getRawResult () {
 return res;
 }
 public boolean exec() {
  if (level == 1) { // Trivial case: 1 × 1 matrix
  res = matrix [0] [0];
  } else if (level == 2) { // Base case: 2 × 2 matrix
  res = matrix [0] [0] * matrix [1] [1]– matrix [1] [0] *
  matrix [0] [1];
  } else { // N × N matrix
  res = 0;
   // have a list of tasks to execute at a later time
   final List < ForkJoinTask < Double >> tasks = new
LinkedList < ForkJoinTask<Double >> ();
  for (int j1 = 0; j1 < level; j1++) {
    final double [] [] m = new double[level – 1] [];
 for (int k = 0; k < (level – 1); k++)
   m[k] = new double [level – 1];
for (int i = 1; i < level; i++) {
 int j2 = 0;
for (int j = 0; j < level; j++) {
 if (j == j1)
    continue;
  m[i – 1] [j2] = matrix [i] [j];
  j2++;
 }
}
final int idx = j1;
tasks.add (new ForkJoinTask < Double > () {
   double result;
   public Double getRawResult () {
     return result;
   }
   protected void setRawResult (Double value) {
     result = value;
  }
  protected boolean exec () {
    result = Math.pow (–1.0, 1.0 + idx + 1.0) *
   matrix [0] [idx] * pool.invoke (new DeterminantTask (m, level – 1));
   return true;
   }
 });
}
// once the task is done completing all of the additional
tasks, it will add all of the results.
    for (final ForkJoinTask < Double > task :invokeAll(tasks)) {
         res += task.getRawResult ();
       }
   }
   return true;
}
```

Figure 2. Parallel algorithm of Laplace expansion for finding determinants

However, after designing the parallel algorithm for finding determinants using Laplace expansion, we have moved forward to test the sequential version of LU Decomposition shown in [2], which is significantly better in complexity.

Then, We have designed our parallel version of LU Decomposition, as shown in Figure 3. The parallelism is achieved by calculating the pivot for each column simultaneously. Due to how the decomposition works, it seems impossible to parallelize the decomposition. It requires going in order, which doesn't fit in parallel processing design schemes, whereas the pivot code could be run in parallel because the pivots have nothing to do with each other.

Hence, each thread calculates the pivot, which involves only calculating numbers close enough to the actual numbers.

For each column, it is calculating the pivot, and since the pivot does not rely on the previous element for proper calculation, it is possible to make it parallel. It works by going through the matrix diagonally and tries to get the next highest column on the current row up until the last row.

```
public boolean exec () {
    final AtomicBoolean sign = new AtomicBoolean (true);
   // init permutation
  for (int i = 0; i < matrix.length; ++i)
    permVector [i] = i;
   final List < RecursiveAction > tasks = new LinkedList < RecursiveAction > ();
  for (int jIdx = 0; jIdx < matrix.length – 1; ++jIdx) {
     final int j = jIdx;
    tasks.add (new RecursiveAction () {
     protected void compute () {
     // Find pivot element in the j-th column.
    double max = Math.abs (matrix [j] [j]);
    int i_pivot = j;
    for (int i = j + 1; i < matrix.length; ++i) {
     double aij = Math.abs(matrix [i] [j]);
     if (aij >max) {
      max = aij;
       i_pivot = i;
      }
     }
   if (i_pivot != j) {
       // because this is a multiple action block, it must be
      synchronized – swap pivots
      synchronized (tasks) {
     double [] swap = matrix [i_pivot];
    matrix [i_pivot] = matrix [j];
   matrix [j] = swap;
  }
  // same with this
  synchronized (sign) {
  // update permutation
  int swap2 = permVector [i_pivot];
  permVector [i_pivot] = permVector [j];
  permVector [j] = swap2;
  synchronized (sign) {
     sign.set (!sign.get ());
     }
   }
  }
 }
 });
}
invokeAll (tasks);
for (int j = 0; j < matrix.length – 1; ++j) {
  double pivot = matrix [j] [j];
if (pivot != 0.0) {
   // calculate decomposition based on the pivot if not 0
  for (int i = j + 1; i < matrix.length; ++i) {
     final double v = matrix [i] [j] / pivot;
    matrix [i] [j] = v;
    for (int k = j + 1; k < matrix.length; ++k)
     matrix [i] [k] –= v * matrix [j] [k];
    }
   } else {
     throw new ArithmeticException ("pivot can't be 0,aborting");
   }
 }
 determinant = sign.get () ? 1 : –1;
// go through the decomposed matrix diagonally and multiply to find the determinant
for (int i = 0; i < matrix.length; ++i)
   determinant *= matrix [i] [i];
  return true;
}
```

Figure 3. Parallel algorithm of LU Decomposition for finding determinants

4. Analysis

In order to implement and test our algorithms, we have used the software/hardware specified in Table 1. However, The first execution of all tests is always slower due to the JIT nature of the JVM, which is why it speeds up after further executions. Hence, for consistent results, we have ran every test t_i multiple times (i.e. T times), excluded the first run, and taken the average execution time of the remaining tests:

$$Run\ Time = \frac{\sum_{i=2}^{n} t_i}{T-1}$$

The sequential run time T_s is the time elapsed between the start and end of the execution of the program, while the parallel run time T_p is the time taken from the execution of the first processor to the end of the last processor in a machine with p processors.

To analyze results, some metrics are used, which are, Speed-up and Efficiency. Speed-up is given by the ratio of the sequential time to the parallel time, and measures how much faster the algorithm became after parallelization. Efficiency are given by the ratio of the speed-up to number of processors, and measures how good utilization is achieved by the parallel solution. The general formulas of Speed-up and Efficiency are given below:

$$Speed\text{-}up\ S = \frac{Sequential\ Run\ Time}{parallel\ Run\ Time} = \frac{T_s}{T_p}$$

$$Efficiency\ E = \frac{Speed\ up}{Processors} = \frac{S}{p} = \frac{T_s}{pT_p}$$

We have also defined a Threshold, which is, a point where the parallel run time becomes faster than the sequential time, that is, the parallel version becomes worthy to use against the sequential one.

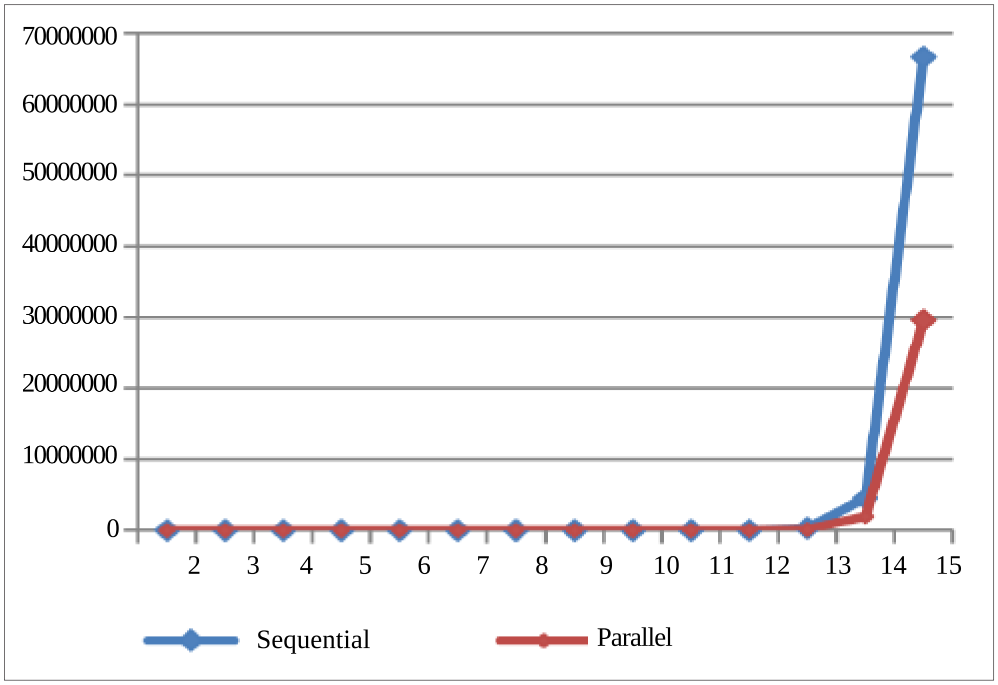

Figure 4. Sequential and parallel run time for Laplace method

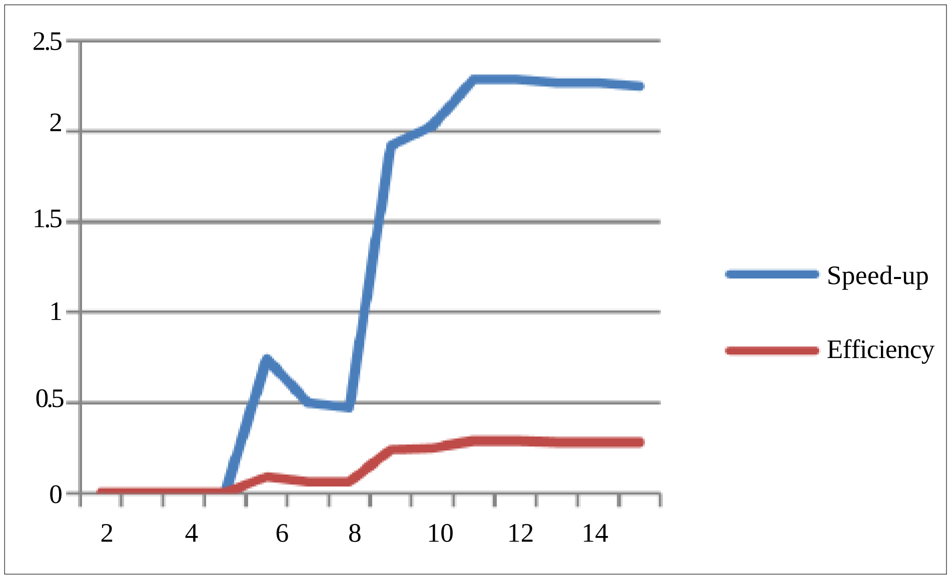

Figure 5. Speed-up and efficiency for Laplace method

Software/Hardware	**Specification**
CPU	Intel Core i7-3770 CPU @ 3.40GHz
Memory	16GB
IDE Eclipse	SDK Version 4.2.2
JVM	JavaSE-1.7 (JVM 7)
Platform	Windows 7 Professional 64-bit

Table 1. Hardware/Software Specifications

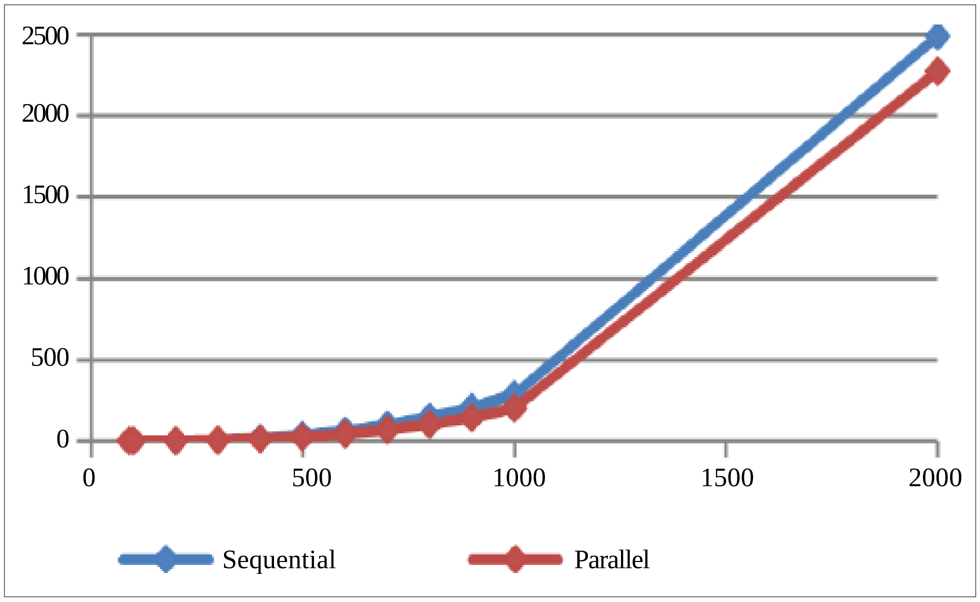

Figure 7. Sequential and parallel run time for LU Decomposition method

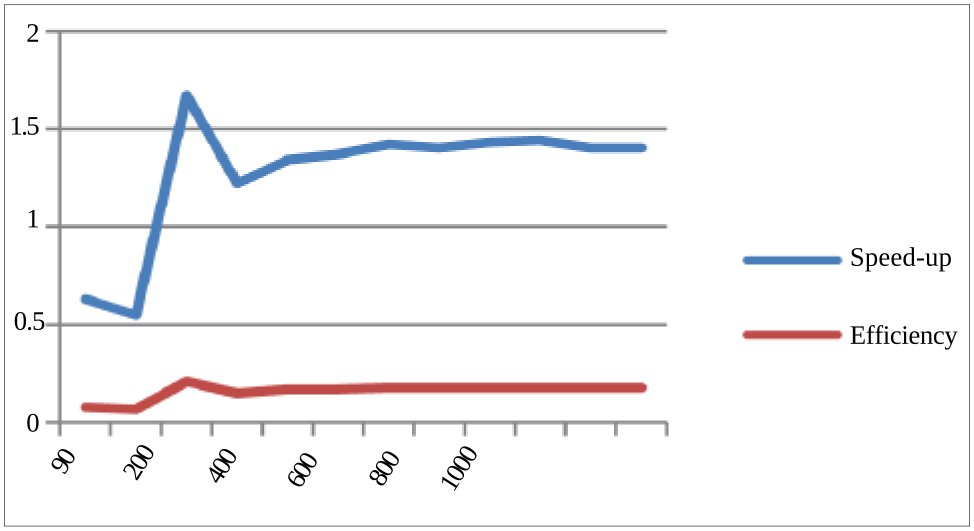

Figure 8. Speed-up and efficiency for LU Decomposition method

4.1 Laplace expansion

The test results for Laplace expansion method for both sequential and parallel algorithms are shown in Table II and Figure 4. The trivial $N = 1$ case is omitted, and the other $N < 6$ cases are shown although the results are roughly zero. The test is done until $N = 15$ due the time-consuming process of calculating the average of multiple runs of each matrix size for both sequential and parallel.

As shown in Table 2 and Figure 4, the threshold is $N = 8$. The sequential run time has provided better results only with small N values (i.e. $N < 8$). Beginning from $N = 8$ and above, the parallel run time is much faster. At maximum, the parallel time reached 129% faster than the sequential time.

The speed-up and efficiency tend toward stability over time at 2.2 and 0.28, respectively, as shown in Figure 5.

As shown in Figure 6, the results are printed for both algorithms, along with the determinant value. We always take the average

N	**Execution Time in Milliseconds**		**Performance Metrics**	
	Sequential	***Parallel***	***Speed-up***	***Efficiency***
2	0 (roughly)	0.08	0	0
3	0 (roughly)	0.03	0	0
4	0 (roughly)	0.08	0	0
5	0 (roughly)	0.24	0	0
6	0.7	0.94	0.74	0.09
7	2.77	5.51	0.50	0.06
8	2.37	4.99	0.47	0.06
9	19.8	10.3	1.92	0.24
10	184.99	91.23	2.03	0.25
11	2063.63	902.02	2.29	0.29
12	24762.4	10833.97	2.29	0.29
13	320659.79	141036.4	2.27	0.28
14	4497124.69	1981112.2	2.27	0.28
15	66655546	29590003	2.25	0.28

Table 2. Test Results for Laplace Expansion Method

```
Enter dimension of matrix: 12
Parallel Run:
Test 1: 10964.2ms
Test 2: 10852.4ms
Test 3: 10845.4ms
Test 4: 10836.0ms
Test 5: 10834.1ms
Test 6: 10830.7ms
Test 7: 10830.4ms
Test 8: 10827.8ms
Test 9: 10829.3ms
Test 10: 10819.6ms
The determinant is –1.699823947469541E22

Sequential Run:
Test 1: 24430.7ms
Test 2: 24713.2ms
Test 3: 24785.9ms
Test 4: 24791.0ms
Test 5: 24753.5ms
Test 6: 24761.7ms
Test 7: 24761.3ms
Test 8: 24754.9ms
Test 9: 24762.7ms
Test 10: 24777.4ms
The determinant is –1.699823947469541E22
```

Figure 6. Sequential and parallel run time for Laplace method

run time excluding the first run.

4.2 LU Decomposition

The test results after applying LU Decomposition for both sequential and parallel are shown in Table III. Due to the efficiency of LU Decomposition compared to Laplace expansion method, we have omitted all results where $N < 90$ because their run time is roughly zero. The test results are shown from $N = 90$ to $N = 2000$, which is a huge number for a matrix.

N	**Execution Time in Milliseconds**		**Performance Metrics**	
	Sequential	***Parallel***	***Speed-up***	***Efficiency***
90	0.5	0.8	0.63	0.08
100	1.33	2.44	0.55	0.07
200	2	1.2	1.67	0.21
300	7.2	5.9	1.22	0.15
400	17.5	13.1	1.34	0.17
500	34.2	25	1.37	0.17
600	62.5	43.9	1.42	0.18
700	98.2	70.1	1.4	0.18
800	147	102.65	1.43	0.18
900	208.77	145.35	1.44	0.18
1000	285.44	203.18	1.4	0.18
2000	2490.2	1778.8	1.4	0.18

Table 3. Test Results for LU Decomposition Method

As shown in Figure 7, our threshold is $N = 200$, where the parallel time takes over and continues to be better than the sequential time for the rest of our experiment. The maximum ratio of how fast is the parallel version compared to the sequential have reached 44%.

The speed-up and efficiency of this method are shown in Figure 8. They tend toward stability over time, at 1.4 and 0.18, respectively.

5. Conclusion and Future Work

In this paper, we have applied two methods to find the determinants of NxN matrices using both sequential and parallel algorithms. The new parallel algorithm of Laplace expansion provided much better results than the sequential algorithm, it achieved 2.29 speed-up (i.e. 129% faster) at maximum. The other new parallel algorithm of LU Decomposition provided better results than its sequential version, it reached 1.44 speed-up, that is, 44% faster than the original sequential one.

In future, we plan to parallelize other Gaussian Elimination methods like Gauss-Jordan Elimination. We also aim to formalize our experiments.

6. Acknowledgement

The authors are pleased to thank Adel Almalki on his careful reading and valuable comments, which improved the research clarity and conciseness.

References

[1] Lea, D. (2000). A Java fork/join framework, *In*: Proc. ACM 2000 conf. Java Grande, p. 36–43.

[2] The MathWorks and the National Institute of Standards and Technology (NIST). (2012, Nov. 9) *JAMA: A Java Matrix Package* (Version 1.0.3) [Online]. Available: http://math.nist.gov/javanumerics/jama.

[3] Rote, G. (2001). Division-Free Algorithms for the Determinant and the Pfaffian: Algebraic and Combinatorial Approaches, *In*: Computational Discrete Mathematics, Berlin, Germany: Springer Berlin Heidelberg, p. 119-135.

[4] Salihu, A. (2012). New Method to Calculate Determinants of n×n (ne ≥ 3) Matrix, by Reducing Determinants to 2nd Order, *International Journal of Algebra*, 6 (19) 913–917, Apr.

[5] Hardwick, J. C. (1997). Practical Parallel Divide-and-Conquer Algorithms, Ph.D. dissertation, School of Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA.

[6] Vavra, L. M. (2010). Cramer's Rule, M.S. dissertation, Dept. Math., Univ. Warwick, Warwick, England.

[7] Borodin, A. *et al.* (1982). Fast Parallel Matrix and GCD Computations, *Information and Control*, 52 (3) 241-256, Mar.

[8] Mahmood, A. *et al.* (1991). A log2 n parallel algorithm for the determinant of a tridiagonal matrix, *In*: IEEE Int. Sympoisum on Circuits and Systems, Singapore, p. 3043–3046.